The University of Chicago
Department of Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2007
Lab #3 (08/08/2007)

Name:			
Student ID:		*Lab Instructor:*	Borja Sotomayor

Do not write in this area

1	2	3	4	5	**TOTAL**

Maximum possible points: 30+ 10

In this lab, your exercises must be compiled using **`make`***. In particular, the instructor must be able to compile your exercises simply by running* **`make exN`** *(where N is the exercise number; e.g.,* **`make ex3`** *to compile exercise 3). You are provided with a sample Makefile which you can use as a starting point.*

FLTK: A simple GUI library

In the first part of the lab you will be writing a series of C/C++ programs that use the FLTK library (http://www.fltk.org/), a simple library for building graphical user interfaces (GUI). This library will enable you to write programs that open windows, react to events like button clicks, etc. The FLTK library is already installed on the CS machines, so there is no need to install it first.

Exercise 1 <<5 points>>

In this first exercise, you are provided with C/C++ code that uses the FTLK library. The goal of this exercise is for you to see how the library is included in your program, and how this affects how your program is compiled. In the next exercise you will build on this code.

You do not need to hand in any code for this exercise. When you are done, raise your hand and the instructor will verify that you've compiled and run the FLTK program correctly.

The code you will compile and run is similar to the example included in the FLTK documentation:

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv)
{
	Fl_Window *window = new Fl_Window(300,200);
	Fl_Box *box = new Fl_Box(20,20,260,100,"Hello, World!");
	box->box(FL_UP_BOX);
	box->labelsize(36);
	box->labelfont(FL_BOLD+FL_ITALIC);
	box->labeltype(FL_SHADOW_LABEL);
	window->end();
	window->show(argc, argv);
	return Fl::run();
}
```

For now, don't worry about the code inside the main function. Nonetheless, notice the **#include** statements at the top of the file. These statements include the FLTK header files, so the compiler will be aware of the classes, functions, and other declarations included with the FLTK library. Since the FTLK library is already installed on the CS machines, the include files are already in a well-known standard location (in **/usr/include**; take a look at this directory, and you will see header files from a wide variety of libraries). So, the include statement specifies the path to the header files <between less-than and greater-than symbols>. This instructs the compiler to search for the files in standard locations (like **`/usr/include`**).

Now, try to compile the program:

```
g++ test_fltk.cpp -o test_fltk
```

You should see a lot of "undefined reference" errors like this:

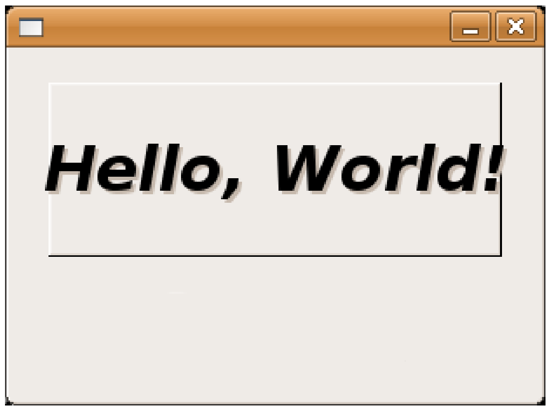
```
/tmp/ccwUroJt.o: In function `main':
test_fltk.cpp:(.text+0x42): undefined reference to `Fl_Window::Fl_Window[in-
charge](int, int, char const*)'
test_fltk.cpp:(.text+0x12c): undefined reference to
`fl_define_FL_SHADOW_LABEL()'
test_fltk.cpp:(.text+0x146): undefined reference to `Fl_Group::end()'
...
```

These are all *linker errors*, meaning that the program compiled successfully, but failed in the linking step because the compiler could not find the implementation of the FLTK functions used in our program (in fact, if you try to compile without linking, using "**`g++ -c test_fltk.cpp -o test_fltk.o`**", you will see no error messages at all).

What is the problem? Although we have correctly included the FLTK headers in our code, and those headers are correctly installed on our systems, we still have to tell the compiler to link with the FLTK libraries. This is done with the "**`-l`**" option:

```
g++ test_fltk.cpp -l fltk -o test_fltk
```

Now, your program should compile fine. If you run it, you should see the following:



Now, let's take a closer look at the code inside the main function:

```
Fl_Window *window = new Fl_Window(300,200);
Fl_Box *box = new Fl_Box(20,20,260,100,"Hello, World!");
```

First of all, we create a new window (of size 300 pixels by 200 pixels). This is done by creating a new **Fl_Window** object. Next, we create a "box" with some text in it. The box is created in the window we just created and, in particular, it is placed in coordinates (20,20). In most GUI libraries, the origin point of a window is its upper-left corner. The box has size 260 pixels by 100 pixels, and its text is "Hello, World!". As we can see when running the program, this box appears at the top of our window.

```
box->box(FL_UP_BOX);
box->labelsize(36);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labeltype(FL_SHADOW_LABEL);
```

The elements in our GUI (windows, boxes, buttons, etc.) are usually called *widgets*. Widgets generally have a set of attributes that we can modify, to customize the look of our GUI. So, after creating the box, we modify a couple of its attributes using member functions of the **FL_Box** class:

- We indicate the type of box using the box() method. **FL_UP_BOX** indicates that this is a "raised" box (notice how the box stands out from the rest of the window).
- We set the font size of the text in the box using the **labelsize()** method.
- We set the font properties (bold and italic) using the **labelfont()** method.
- We set the type of label using the **labeltype()** method. In this case, by specifying **FL_SHADOW_LABEL** we have added a shadow effect to the text.

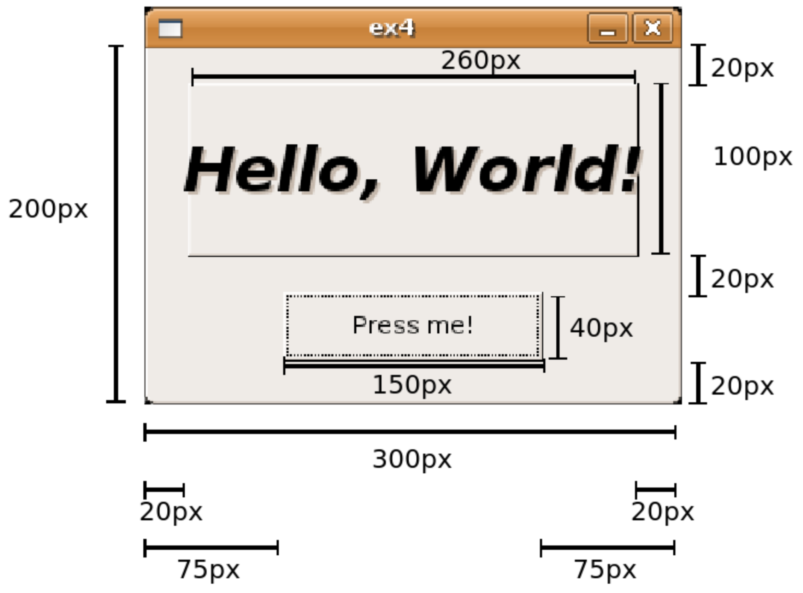
```
window->end();
window->show(argc, argv);
return Fl::run();
```

Finally, we indicate that we have finished specifying this window using the **end()** method. If we created more widgets, they would not be added to our window. Next, we make the window visible using the **show()** method. Finally, we call FLTK's **run()** function, which will create and visualize the GUI we just specified.

Exercise 2 <<10 + 5 points>>

The previous exercise showed a very simple example of a FTLK-based program. There are dozens of other widgets we could add to our program, and literally hundreds of other aspects we could modify in our GUI, and the example is just meant to illustrate how we can add new functionality to our programs (like a GUI) with relatively little effort (compared to writing a GUI library ourselves). If you want to learn more about FLTK, the best starting points are the FTLK website and the FLTK programming manual (http://www.fltk.org/doc-1.1/toc.html). In fact, in this exercise, you will have to look at the manual to find instructions on how to add a new widget to your program.

In this exercise, you must add a *button* widget, with the text "Press me!" on it. The dimensions and position of the button are specified in the following figure:



Note that adding this button involves adding a single additional line to your program.

Exercise 3 <<Extra credit: 5 points>>

Modify the program so that, when the button is clicked, its label changes to "You clicked me!".

GSL: GNU Scientific Library

In this part of the lab you will install a library that is not currently installed on the CS machines. This library is the GNU Scientific library, a numerical library for C/C++ that provides advanced mathematics capabilities not included in the standard C/C++ math libraries.

Exercise 4 <<5 points>>

In this exercise you will install the GSL on your machine, and will then compile and run a sample program. You do not need to hand in any code for this exercise. When you are done, raise your hand and the instructor will verify that you've compiled and run the GSL program correctly.

First of all, download the GSL library from the GSL website. You can use the following link: ftp://ftp.gnu.org/gnu/gsl/gsl-1.9.tar.gz. Save the file to a temporary directory, like **/tmp** or **/var/tmp**. Note that, from the command line, you can use the **wget** command to download a file from an HTTP or FTP server:

```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-1.9.tar.gz
```

This file is compressed, and we will have to uncompress it. We can do this with the "**tar**" program:

```
tar xvzf gsl-1.9.tar.gz
```

This will result in a new directory called **gsl-1.9**. Change to that directory:

```
cd gsl-1.9
```

There is no universal way of installing a new library, so we must generally look for an **INSTALL** or **README** file in the downloaded library with instructions on how to install the library. It is also common to find installation instructions on the software's website. In GSL's case, there is an **INSTALL** file. You do not have to read through it, since installation instructions are included in this handout.

Whenever we download a new library, the installation is usually divided into three steps: configuring the installation, building the library binaries from the source code, and actually installing the library. There are more automatic ways of doing this, and many GNU/Linux distributions provide package management systems (like APT, RPM, ...) that will install additional libraries with much less hassle (however, those systems are beyond the scope of this course). In general, the best starting point is always the installation instructions provided by the authors of the software we want to install (e.g., in their website, in the INSTALL/README file, etc.)

So, first of all, we need to configure the installation. This is accomplished by running a "**configure**" script that is already provided for us:

```
./configure --prefix=$HOME
```

The "**--prefix=$HOME**" option simply specifies that we want to install the software in our home directory. The default is to install the software in well-known standard directories (e.g., **/usr/local**). However, we do not have administrative access on the CS machines (i.e., we would not be able to place new files in directories like **/usr/local**), so we must install GSL in our home directory.

Once you run the above script, you should see several messages indicating if we meet all the prerequisites to install the library. If we don't meet the prerequisites (e.g., we're missing a required library), the configure script will tell us about it. In our case, the configure script will terminate without errors.

Next, we have to build the library. Take into account that we are provided with the source code of the library, and we will not be able to use it until we've compiled it. In some cases, we can download pre-compiled binaries of a library and skip this step.

To build the library, the only thing we need to run is the following:

```
make
```

Your computer will spend about five minutes compiling the library. Once it is done, you can install the library by running the following:

```
make install
```

If you look at your home directory, you will notice new directories like "**lib**", "**include**", etc.

Now, we are ready to try out a sample program. We will use the one provided in the GSL's Reference Manual (http://www.gnu.org/software/gsl/manual/):

```
#include <iostream>
#include <gsl/gsl_sf_bessel.h>
using namespace std;
```

```
int main (void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0 (x);
    cout << "J0(" << x << ") = " << y << endl;
    return 0;
}
```

This program computes the value of the Bessel function $J_0(x)$ for x=5. Notice how we're including the GSL's Bessel header:

```
#include <gsl/gsl_sf_bessel.h>
```

And then using the function **`gsl_sf_bessel_J0()`** in our program.

However, compiling this program won't be as simple as in the FTLK example. Try the following:

```
g++ test_gsl.cpp -o gsl
```

You will see the following error messages:

```
test_gsl.cpp:2:31: gsl/gsl_sf_bessel.h: No such file or directory
test_gsl.cpp: In function `int main()':
test_gsl.cpp:7: error: `gsl_sf_bessel_J0' undeclared (first use this function)
test_gsl.cpp:7: error: (Each undeclared identifier is reported only once for
   each function it appears in.)
```

The compiler can't find the GSL header file and, consequently, doesn't know what the **`gsl_sf_bessel_J0`** function is. The reason for this is that we have installed the GSL library (including its header files) in a non-standard location (our home directory). So, when compiling, we must inform the compiler of where those header files can be found:

```
g++ test_gsl.cpp -o gsl -I$HOME/include
```

However, we will still be unable to compile the program:

```
/tmp/ccigPHDq.o: In function `main':
test_gsl.cpp:(.text+0x20): undefined reference to `gsl_sf_bessel_J0'
collect2: ld returned 1 exit status
```

Although the compiler can find the header files (and thus knows what the **`gsl_sf_bessel_J0`** function is), it can't find the implementation for that function. Similarly to the FLTK example, we need to link with the GSL libraries:

```
g++ test_gsl.cpp -o gsl -I$HOME/include -lgsl -lgslcblas
```

After running the above, you should see the following message:

```
/usr/bin/ld: cannot find -lgsl
collect2: ld returned 1 exit status
```

We've specified that we want to link with the GSL libraries, but the compiler can't find them. Once more, this is due to the fact that the libraries are in a non-standard location (our home directory), so we have to tell the compiler where it should search for libraries:

```
g++ test_gsl.cpp -o gsl -I$HOME/include -lgsl -lgslcblas -L$HOME/lib
```

At last, our program will compile! However, we're not out of the woods yet... If we try to run it:

```
./gsl
```

We will get the following error:

```
./gsl: error while loading shared libraries: libgsl.so.0: cannot open shared
object file: No such file or directory
```

The problem is that now the operating system cannot find the GSL library, which it needs to dynamically link with. Again, this is because we installed the library in a non-standard location. In a Linux system, we can specify where the operating system should look for libraries by setting the LD_LIBRARY_PATH environment variable:

```
export LD_LIBRARY_PATH=$HOME/lib
```

Now, if you run the program, you should finally see the following:

```
J0(5) = -0.177597
```

Of course, keeping track of all the options you must specify when running the compiler can be very tiresome (and error-prone). This is an example where using make can be very useful. For example, we could use the following makefile:

```
INCLUDES = -I$(HOME)/include
LDFLAGS = -lgsl -lgslcblas -L$(HOME)/lib
```

```
CPPFLAGS = $(LDFLAGS) $(INCLUDES)

all: test_gsl

test_gsl: test_gsl.cpp

clean:
	rm test_gsl
```

The above Makefile uses make's **`CPPFLAGS`** variable, which allows us to specify what options should be passed along to g++. Notice how it is defined in terms of two other variables: **`INCLUDES`** and **`LDFLAGS`.**

With the above Makefile, compiling the program is as simple as running this:

```
make
```

Exercise 5 <<10 points>>

Find the GSL function that will allow you to evaluate a polynomial (see the GSL Reference Manual: http://www.gnu.org/software/gsl/manual/). ***Use this GSL function*** to write a program that will evaluate the following polynomial:

$$3x^4+5x^2+2x-1$$

For x=1, 2, 3, ... , 9, 10. The output of your program should be:

```
poly(1) = 9
poly(2) = 71
poly(3) = 293
poly(4) = 855
poly(5) = 2009
poly(6) = 4079
poly(7) = 7461
poly(8) = 12623
poly(9) = 20105
poly(10) = 30519
```

Exercise 6 <<Extra credit: 5 points>>

Modify the above program so that the user can specify an arbitrary polynomial of any degree. The user must also be able to specify the range of values to evaluate: a starting value, and end value, and an increment. For example, in the previous exercise, the starting value was 1, the end value was 10, and the increment was 1.